Building a FIWARE Smart City Platform

Peter Salhofer
FH JOANNEUM
peter.salhofer@fh-joanneum.at

Julia Buchsbaum
FH JOANNEUM
julia.buchsbaum@edu.fh-joanneum.at

Michael Janusch
FH JOANNEUM
michael.janusch@edu.fh-joanneum.at

Abstract

This paper describes the architecture of a comprehensive IoT solution entirely based on the FIWARE platform. The application is designed to record data from environmental sensors and to eventually visualize them on a Smart City Dashboard. Besides solving certain architectural and technical issues, one particular challenge arose from the fact that some of the sensors were assumed to be mounted on public transportation vehicles like buses and trams. It could be shown that the FIWARE platform provides a range of components that allows for building such an IoT platform in a very efficient way.

1. Introduction

FIWARE is the result of several EU funded projects with the goal to provide a a set of standardized APIs supporting the creation of Smart Applications in various fields [1][2][3][4]. Currently FIWARE is intensively promoted by the so called FIWARE Foundation[5] that tries to push the take-up of the FIWARE stack. This stack consists of a broad set of APIs as well as reference implementations of these API, resulting in a huge set of modularized, open-source software components that are grouped in so called general enablers.

This paper presents the results of a project that implemented an IoT solution exclusively based on these FIWARE components. It is therefore structured in the following way:

- Section 2 describes the use case that was implemented
- Section 3 briefly discusses those FIWARE components that have been used to solve the use case scenario
- Section 4 presents the architecture of the final solution
- Section 5 summarizes our findings

2. The Problem Statement

The solution presented in this paper was a prototype for an actual IoT installation that will be realized over the next couple of months. Thus, while implementing the software, sensors were not in place but were represented either by mockups or by actual makeshift sensors based on a raspberry PI. The general idea was to collect data from various sensors that are mounted throughout a city area. These sensors were supposed to collect the following data:

- Temperature
- Humidity
- Concentration of fine particles (PM2.5 and PM10)

All sensor data needs to be stored in a data sink and the application should provide an easy-to-use dashboard, visualizing the data using geographical maps, tables and different charts. One long-term goal was to use the collected data combined with external data (e.g. regional weather data) to generate a prediction model for the particulate matter concentration, motivated by its enormous impact on the health of the population[6].

The plan was to have some of these sensors mounted on predefined points that were suggested by the environmental department. In order to have the necessary power supply, traffic lights or light poles closest to these points were chosen. On the other side some sensors shall be mounted on public transportation vehicles (busses and trams). These mobile sensors also need to report their current position along with the other sensor data. In addition, data transmission should be performed in a way, so that there will be a new sample every 200 meters.

3. The FIWARE Platform

The core of the FIWARE ecosystem is the so called FIWARE platform. It is a set of public and free-to-use API specifications that come along with open source reference implementations.

The FIWARE platform is grouped in seven major parts called the “generic enablers (GEs)”[7]. Every GE represents a certain aspect of FIWARE services and also provides one or more components along with reference implementations that support the specified APIs. Additionally, there are so called “domain specific enablers (DSEs) that (will) provide components for certain domains like health, energy and so on. The general enablers are organized as follows:

URI: https://hdl.handle.net/10125/60175
ISBN: 978-0-9981331-2-6

- **Data/Context Management**: This contains all components that are needed to store, access, process and analyze data as part of a smart application
- **Internet of Things (IoT) Services Enablement**: Here are all components needed to setup sensor networks and routing sensor data to other GEs.
- **Advanced Web-based User Interface**: Components to design user interfaces, including geographical information and interactive 3D charts
- **Security**: Components to add, define and enforce declarative security
- **Advanced middleware and interfaces to Network and Devic**es
- **Applications/Services and Data Delivery**: Components and tools for data visualization, easy generation of mashups and app-store-like distribution of services and data
- **Cloud Hosting**: Components and tools aiming at providing and managing FIWARE services via cloud infrastructure

FIWARE used a great variety of different programming languages (C++, Java, Python, NodeJS, ...) and environments for developing their reference implementations. Fortunately, the FIWARE community provides docker[8] images for every component, which makes dealing with different runtime requirements relatively easy.

In order to get a basic understanding of the components that were required to get the use-case implemented, we will briefly describe them in the following sub-sections.

3.1. The Context Broker - Orion

Probably the most essential API within the entire FIWARE stack is called "Context Broker" and its reference implementation is called Orion[1]. It is a persistent data store with a REST API and therefore could probably be compared to CouchDB[9]. In fact, it is using an instance of MongoDB[2] as its internal data store and offers RESTful access to it via the Open Mobile Alliance's *Next Generation Service Interface* (NGSI) protocol[10]. Since the underlying datastore is a NoSQL document store, also Orion does not use database schemas and allows for the creation of any type of entity. It supports a simple, URL-based query language that also provides projections and pagination. Thus, in cases where a longer list of only a sub-set of attributes is needed, this can be easily achieved. This makes Orion a perfect backend for single-page-applications.

Besides this, Orion supports multi-tenancy via a simple header-field identifying the required tenant. The most important feature, at least when it comes to IoT applications, is Orion's capability to easily subscribe for changes in the data store. In fact, publish-subscribe is the single most important interaction pattern used by the various FIWARE IoT key-components. Subscriptions can be made for specific types of entities (e.g. all Buses), for specific attributes of these entities (e.g. get me informed whenever the 'location' attribute of any bus changes) or for individual entity/attribute combinations (e.g. get me informed once the 'temperature' in Bus25 changes).

3.2. Backend Device Management - IDAS

For managing the interaction with sensors, FIWARE provides a general enabler called Backend Device Management. Its reference implementation is called IDAS[12] and it provides a REST endpoint with the API required for registering sensors and dealing with their data. Before a sensor can be added to the system, in a first step a so-called *service* needs to be created, which serves as the logical endpoint for a group of sensors. Besides this, every new sensor registered with this service gets its own unique *device_id*. Both (the service and the device id) are part of the URI that is used by the sensor to deliver its measurement results. IDAS also takes care of the routing of all incoming sensor data. In FIWARE all data that is produced by sensors is mapped to attributes of entities. For example, let's assume there is a bus with a sensor mounted to it, that periodically transmits its location, the current temperature, humidity and particle matters concentration. Within FIWARE we can first model the bus as a business object of our application. This will most likely include attributes like license plate, model/make, engine type (diesel, gasoline, natural gas or electricity) and others. When registering a new sensor device, all its values need to be mapped to attributes of a specific entity stored in the context broker. Thus, whenever a sensor sends a new sample, the corresponding attributes of the entity are updated. As a result, whenever we fetch a bus from the context broker, it will also have attributes like location, temperature and so on, that will always contain the latest sensor results.

3.3. Storing Time Series Data - Cygnus

As described in the previous section, IDAS is used to route inbound sensor data to corresponding entity attributes. This makes sure that every entity always represents the most current state of the underlying cyber-physical system. On the other hand, this does not include the availability of historic data, since with

[1] https://github.com/telefonicaid/fiware-orion

[2] https://www.mongodb.com/

every update the previous value of the attribute is overwritten. In order to also keep the previous values of sensor results available, an additional component called Cygnus[13] is required. This component is essentially an extension of Apache Flume[14] that is used to store updates in a persistent storage. It is listening for incoming data that is then forwarded – according to its internal configuration – to one or several data sinks. Possible data sinks are beside others MongoDB, HDFS and PostgreSQL.

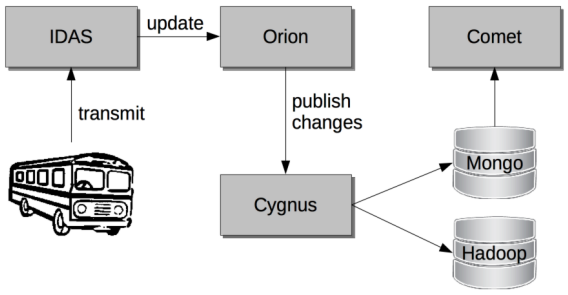

Figure 1: Flow of sensor data within FIWARE

In order to continuously store sensor values over time a subscription with the context broker is created by Cygnus. This will make sure that whenever a particular property of a specific type of entity is changed (e.g. the *location* property of entities of type *bus*) Cygnus receives this information and sends it to the persistence storage (see Figure 1).

This architecture allows for a clear separation of live data stored in the context broker and the historical data stored in any database of choice. Having split the task over several loosely and asynchronously connected components allows for high performance and throughput. Probably most important, this can all be achieved without a single line of programming so far.

3.4. Short Term Historic - Comet

Since everything in FIWARE is about REST-based APIs, there is also a component that allows for RESTful access to the historic data sink. The name of this component is Short Term Historic (STH) and the reference implementation is called Comet[15]. It provides an API for reading historic data produced by the component chain described above, but only supports MongoDB data sinks so far.

3.5. Security

None of the components that have been mentioned so far does support security. Thus, whenever one has access to these services, there are no restrictions on what can be done. This includes the creation, modification and deletion of any data or configuration information within the system. Within FIWARE security is conceptually a separate layer that needs to be put atop the other components. One potential benefit of this architectural decision is that the whole security layer can be replaced by another implementation if needed.

FIWARE's standard security infrastructure is based on OAuth2[16] and consists of the following three components:

- Identity Management (IdM): Within the OAuth2 protocol this component is the authorization server, thus, all client applications have to register with the IdM. It also provides a REST API and a web-based user-interface to create users, roles and permissions.
- Policy Decision Point (PDP): This service provides authorization by deciding whether the current user is allowed to perform a certain action
- Policy Enforcement Point (PEP): This is a proxy server that performs the actual authentication and optional authorization checks in interaction with the other two components

The IdM is the central component of the FIWARE security architecture. Its reference implementation is called Keyrock[17] and it is based on OpenStack Keystone[18], which in turn is an open source implementation of the OpenStack Identity API[19]. It is holding all user information and is a single sign-on service for all components and applications. Thus, applications do not necessarily need to maintain user information (especially no private credentials) and one account can be used for all applications using the platform. It has recently undergone a major refactoring, including additional, yet basic features (like modifying and deleting existing permissions) that had not been present in earlier releases.

The second important security component within FIWARE's security architecture is the Policy Decision Point (PDP) with its reference implementation called AuthZForce[20]. The role of this component is to authorize access to protected resources. Therefore, so called permissions are created using the IdM, which in the most basic form are combinations of http request methods and URIs. For example, such a rule could grant unlimited read access to all entities stored in the context broker by combining "GET" as http verb and "/v2/entities" as URI into a permission. These permissions are sent from the IdM to the PDP and stored there. These rules are encoded in the eXtensible Access Control Markup Language (XACML)[21].

The set of security components is completed by the Policy Enforcement Point (PEP) with its reference implementation called Wilma[22]. The PEP is a very simple proxy server that is placed in front of the service

that should be restricted and is acting as the actual *resource server* according to OAuth2. Thus, instead of allowing direct access to a service like Orion, IDAS or Comet, the client application needs to interact with the PEP proxy instead. The actual authentication and authorization flow is shown in Figure 2. Before a protected resource can be accessed, the client application has to get an access-token by logging into the system using an IdM account. The PEP proxy checks for the existence of this token and then for the validity by querying the IdM. If the token is valid, the PEP proxy also gets the name of the current user and a set of roles, that could also be used for full-custom application layer security. This information is cached at the PEP. After this, the PEP makes a query to the PDP to figure out, whether this request is authorized. Only if the PDP agrees, the original request is sent to the protected resource.

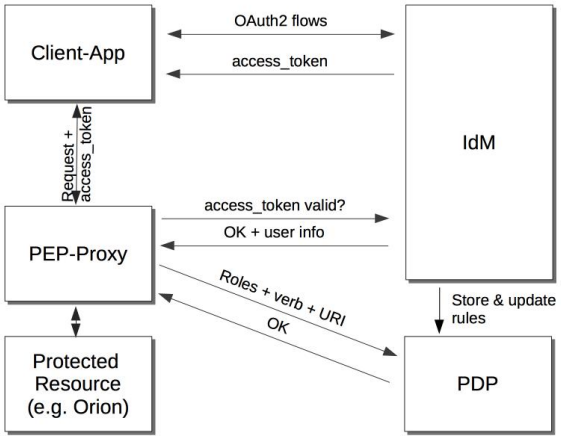

Figure 2: The basic authorization flow[22]

It is important to note, that the PEP has to register and authenticate itself with the IdM. For every client application that is registered with the IdM only one set of PEP credentials is available. Thus, if more than one resource needs to be accessed by the client app in a protected way, multiple PEP proxies are required that either share the same set of credentials or are dedicated to multiple (logical) applications that are registered with the IdM. On the other side, there is a very recent feature called "trusted apps", that allows one PEP to be used by different applications.

3.6. Wirecloud

One of the central aspects of our project was to provide a simple user interface via a web-based dashboard that should allow for:

- administrating the IoT platform (creating, updating, deleting entities, adding/removing sensors, creating/deleting subscriptions)
- visualizing the data using tables, gauges, maps and all sorts of diagrams

The final solution should be easily adaptable to different needs. For this purpose, the FIWARE eco-system provides a component called Wirecloud[23]. It is a web interface that allows for combining small building blocks called widgets or operators into dashboards in a very intuitive, easy to use way. Widgets are components that represent information graphically to the user, whereas operators provide functionality like reading data from other FIWARE services, transforming this data if necessary and pushing it to other components like widgets. Widgets and operators are technically small Javascript applications that come in zipped files with the extension ".wgt". These components can be shared - either for free or for a fee - via the FIWARE Store[3], which is a very valuable resource providing lots of useful building blocks. It is also possible to share complete mashups (pre-configured networks of widgets and operators).

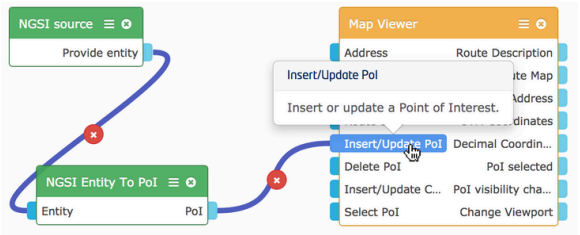

Figure 3: Connecting component using the piping editor[25]

The most convincing feature is the simplicity in building mashups. As shown in Figure 3 widgets and operators can be connected using a drag-and-drop editor. This is enabled by a configuration file that is part of every widget/operator and contains meta-information about input- and output endpoints. Besides this it also exposes configuration properties (e.g. endpoint of the context-broker) that can be defined/changed using a property editor. In this example we see a "NGSI source" operator that is configured to query the context-broker for some entities. The "NGSI Entity to Poi" operator takes each entity and converts it into a POI-object as it is needed by the "Map Viewer" widget. The result of this mashup can be seen in Figure 4.

[3] https://store.lab.fiware.org/

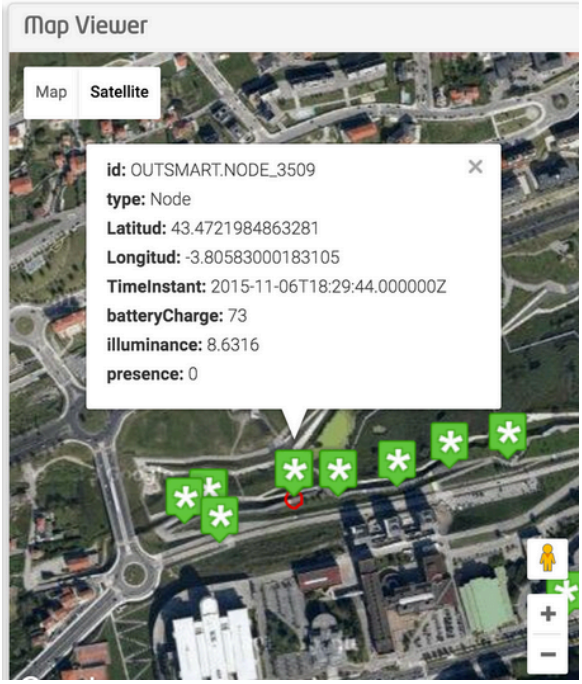

Figure 4: Resulting Mashup[25]

4. The Implementation

In the previous chapters we have given a brief overview of the core components that we have used to implement our IoT platform. Figure 5 shows how these components have been wired together. For the sake of simplicity, the IdM and the PDP along with their interaction with the various PEP proxies have been omitted.

4.1. Security

Wirecloud is a web application that is implemented in python using the Django framework[4]. Out of the box it comes with its own security mechanism, which, however, can easily be re-configured to also support OAuth2 authentication using FIWARE's IdM. This allows for using FIWARE accounts to log into Wirecloud and most operators that are designed to interact with other FIWARE components can be configured to add the OAuth token to their request. So, in our implementation we let these components talk to PEP proxies while having blocked access to the actual FIWARE services. This allows for fine-grained end-to-end security.

Besides securing access to the Wirecloud dashboard and all other FIWARE services used by it, also the south-bound interface to the sensor network is protected. We deliberately used a separate PEP proxy that logically belongs to a different OAuth application. This allows for a clear separation between human users and sensors. Although the IdM allows for the creation of special sensor accounts, these accounts cannot be used in authorization rules. Thus, we are using normal user-accounts also for sensors, allowing for a very restricted access to the IoT platform. Consequently, even if someone gets access to a sensor and its credentials, they can never be used to spoof any other identity, since the credentials used by a sensor only allow for delivering a predefined set of values to a specific endpoint that is exclusively dedicated to a single device.

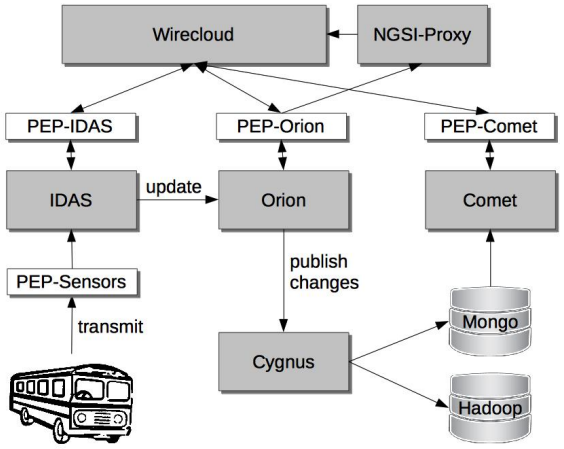

Figure 5: The overall system architecture

4.2. Retrieving Data

As described in section 3, FIWARE strictly separates the current state of the entire system and the historical data. The currents state is always represented by the data stored in the context broker, while historical data can be found in any of Cygnus' data sinks and in the case of MongoDB this data can be easily retrieved using Comet. Thus, to get this information, the context broker needs to get queried using the NGSI protocol. There already exist several off-the-shelf operators that can be retrieved from the FIWARE store. To always get the latest data, however, it is also possible to subscribe with the context broker. In this case the context broker sends updates with every relevant change to the Wirecloud application. Since widgets and operators are written in JavaScript, they are running locally in the client's web browser. To be able to receive the broadcast messages, a so called NGSI-Proxy (see Figure 5) is required. From the context-broker's view, the NGSI-proxy acts as the subscriber and it forwards updates to the actual widget or operator via web-sockets.

Figure 6 shows an example of querying current and historic data. It first fetches a list of known devices (busses, trams and traffic lights with sensors). When the user clicks on an entry in the list, the current values of temperature and humidity are displayed using

[4] https://www.djangoproject.com/

gauges, while the last ten samples are visualized using a line chart. The number of points as well as the chart type can be easily changed, using the corresponding widget's settings dialog.

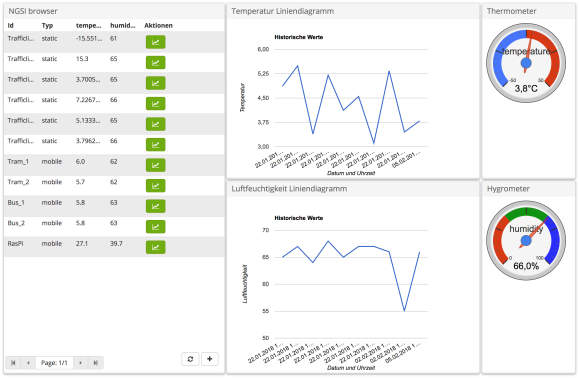

Figure 6: Querying data based in a list of sensors

In Figure 7 we see an alternative approach using a custom widget and operator that was created as part of the project. Instead of rendering entities as a table, they are presented as a form. This allows for selecting an entity (a distinct bus, tram or traffic light), the sensor value of interest (depending on the capabilities of the sensors that report to the selected entity) and a time-range.

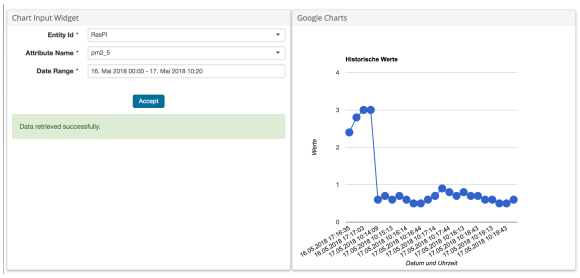

Figure 7: Form-based query of historic data

4.3. Representing Spatial Data

In section 3.6 we have already seen that rendering spatial data on a map is rather simple and straight forward using existing components. Basically, the same approach was used in our project, leading to the result shown in Figure 8. All selected sensors (mobile or stationary) are displayed using different icons representing the type of entity (e.g. bus, tram, light pole, …) . Clicking on an icon will bring up all relevant information about the measuring point. Since the operators connected to this map make use of context-broker subscriptions, every new sensor value is almost immediately displayed on the map. Consequently, all markers representing mobile sensors (busses and trams) are moving over the map in real time. Since there were no such sensors available at the time of development, we have mocked them using a script sending periodic updates at locations along certain points defined in the script.

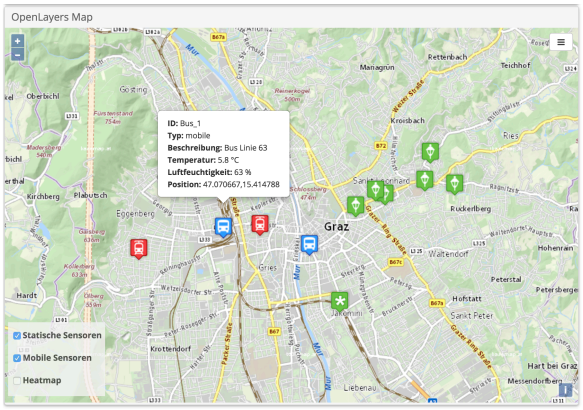

Figure 8: Visualization of the current state

The only change compared to the standard components was the requirement to use a specific[5] open-source map instead of google maps.

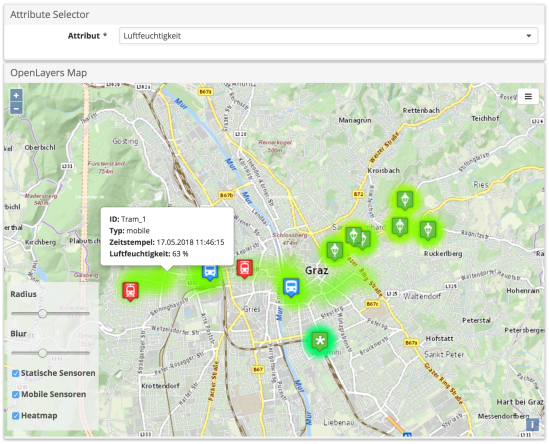

Figure 9: Extending spatial coverage using a heatmap

While visualizing the current state of the system turned out to be straight forward, displaying "historic" data was a bit more demanding. Normally, time series are used to analyze trends over time. In case of mobile sensors (e.g. mounted on delivery trucks), they are often used to track routes. In this case, however, the idea of having mobile sensors was to cover a larger region of the city. Thus, it is not so much the location of the sensor that is of interest, but values of other sensors (e.g. air quality) that had been taken at this location. Analyzing a time window for mobile sensors therefore does not simply reflect changes over time, but

[5] https://www.basemap.at/index_en.html

also expands the area that was covered by these samples within this timeframe.

So, to expand the spatial coverage of our "current state view", we have decided to also allow for including the last 10 samples from any mobile sensor using a heat map. This factor can easily be changed, using the edit dialog of the underlying operator. The result can be seen in Figure 9. Since the location of the mobile sensors is steadily updated, it appears like these sensors were trailing a tail of samples. Every data point is added as a marker, so they can be clicked to reveal all their details. While heatmaps usually only encode the density of their data-points into a color schema, we had to refactor the existing heatmap in order to use the sensor value instead. Thus, an increased density of samples leads to decreased transparency while different sensor values lead to different colors. Since only one of the various sensor values can be used to be encoded by the heatmap, the user can select this value (e.g. temperature, humidity, PM2.5, PM10, ...) using a simple dropdown field. Every sensor value is also mapped to its own color scheme, defining which values should be rendered as low (e.g. blueish), as normal (e.g. greenish) or high (e.g. reddish).

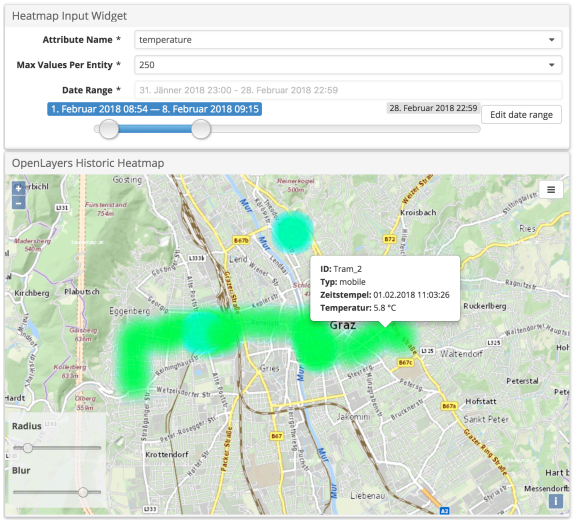

Figure 10: Displaying historic data for a given timeframe

Apart from the question how best to use "historic" samples in order to expand the areal coverage in the "current state" view, a technical issue arose in querying historic data using Comet. In the case of the current state, entities stored in the context-broker have been queried, which come with all sensor data (temperature, humidity, PM2.5, …). Once this data is processed by Cygnus (see section 3.3), every value ends up in an individual document in the database. In the case of the MongoDB sink, there is one collection for every entity. These collections contain documents for every sensor sample consisting of the following properties:

- recvTime: the time this sample was received by IDAS
- attrName: name of value (e.g. "temperature")
- attrType: data type of the value (e.g. "Float", "geo:point", …)
- attrValue: the actual value of that sample

Thus, the only way to correlate those values that belong together is using the timestamp of every entry. Therefore we needed a new Wirecloud operator that queries the last x entries for every sensor type and merges the individual results using their timestamp, so that we can combine a sensor value with the location it was taken at.

Another important view in our dashboard is about representing "real historical" data on a map as it is shown in Figure 10. Here the user can select the sensor type and a timeframe of interest. As already mentioned, the modified heatmap uses the sensor value for color encoding, rather than the density of samples.

4.4. Providing an Administration Interface

Besides representing the data stored in the IoT platform, one goal of the smart city dashboard was to provide means for the administration of the entire system. It turned out, that the Wirecloud eco-system already included most the required functionality. Based on these existing once, custom components have been created as needed.

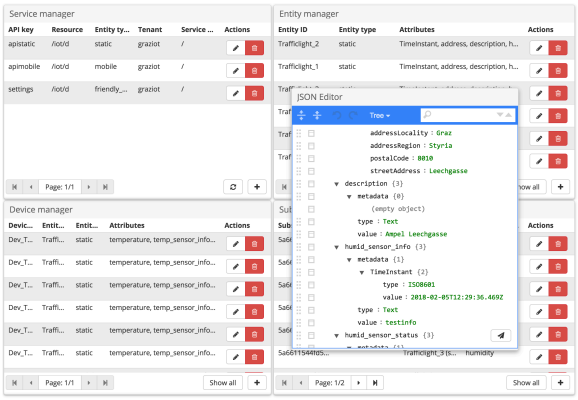

Figure 11: The Admin interface in the dashboard

Thus, we managed to create an admin interface that allows for maintaining all entities, services and devices (see Figure 11). This allows also for the onboarding of new sensors.

5. Conclusions

The idea of the whole project was to figure out, whether it is possible to create a comprehensive, production-class IoT platform using the FIWARE stack with minimal effort. It turned out that all the core components we had to use worked out-of-the-box without any errors. The Wirecloud platform used for implementing the dashboard is based on a well-designed architecture. There is a clear separation between UI-components (widgets) and components to provide and to manipulate data (operators), which greatly improves reusability. All these components can be combined to powerful mashups without a single line of programming using the piping editor. Besides managing connections visually, every single component can be configured using a form-based dialog. The FIWARE store allows for sharing these components and provides hundreds of widgets, operators or pre-configured mashups. Even if the required component cannot be found there, new ones are easily created using simple Javascript. Consequently, the amount of necessary programming was extremely low especially compared to the achieved result.

Another surprising outcome was the resource-consumption and the performance of the whole system. We are hosting our platform in the FIWARE Lab Cloud. The application consists of 15 docker containers that are running on a single virtual machine with two cores and 4GB RAM. Although the data model currently consists of only 11 Entities representing the setting of the planned implementation (2 Buses, 3 Trams, 6 static sensors), there is a relatively high number of services running on a single machine. Nevertheless, no notable latency in working with the system occurs. Of course, there needs to be a thorough analysis of the system, including systematic performance tests, which is going to happen as part of a follow-up project.

Besides this, current activity on the various Github repositories indicates that there is a lively community behind the platform. Thus, based on the results of this project, we can really recommend using the platform for IoT projects.

6. References

[1] Publications Office of the European Union, 2011, *FI-WARE: Future Internet Core Platform*, EU, Brussels, Belgium

[2] Publications Office of the European Union, 2016, *FI-GLOBAL: Building and supporting a global open community of FIWARE innovators and users*, EU, Brussels, Belgium

[3] Publications Office of the European Union, 2017, *A FIWARE-based SDK for developing Smart Applications*, EU, Brussels, Belgium

[4] Publications Office of the European Union, 2017, *Bringing FIWARE to the NEXT step*, EU, Brussels, Belgium.

[5] FIWARE, 2017, https://www.fiware.org/foundation/

[6] Giuliano Polichetti, Stefania Cocco, Alessandra Spinali, Valentina Trimarco, Alfredo Nunziata, 2009, *Effects of particulate matter (PM10, PM2.5 and PM1) on the cardiovascular system*, in Toxicology, Volume 261, Issues 1-2, Pages 1-8

[7] FIWARE 2017, *The FIWARE Catalogue*, https://catalogue.fiware.org/.

[8] John Fink, 2014, *Docker: a Software as a Service, Operating System-Level Virtualization Framework*, code{4}lib Journal, Issue 25, 21.04.2014

[9] Jan Lehnardt, Noah Slater, J. Chris Anderson, 2010, *CouchDB: The Definitive Guide, O'Reilly Media, Inc.*

[10] Open Mobile Alliance, 2012, *NGSI Context Management, http://www.openmobilealliance.org/release/NGSI/V1_0-20120529-A/OMA-TS-NGSI_Context_Management-V1_0-20120529-A.pdf.*

[11] FIWARE Orion Team, 2017, *FIWARE-ORION Documantation – Entity service paths*, http://fiware-orion.readthedocs.io/en/master/user/service_path/index.html

[12] Carlos Ralli Ucendo, 2016, *Backend Device Management – IDAS*, https://catalogue.fiware.org/enablers/backend-device-management-idas

[13] FIWARE Cygnus Team, 2017, *Cygnus*, http://fiware-cygnus.readthedocs.io/en/1.2.2/index.html

[14] Apache Flume Team, 2017, *Apache Flume User Guide*, https://flume.apache.org/FlumeUserGuide.html

[15] FIWARE Comet Team, 2016, *FIWARE Short Time Historic (STH) - Comet documentation* ,https://fiware-sth-comet.readthedocs.io/en/latest/

[16] D. Hardt (Ed.), 2012, *The OAuth 2.0 Authorization Framework*, https://tools.ietf.org/html/rfc6749

[17] Joaquín Salvachúa and Álvaro Alonso, 2016, *Identity Management – KeyRock*, https://catalogue.fiware.org/enablers/identity-management-keyrock

[18] The OpenStack Foundation, 2017, *Keystone, the OpenStack Identity Service*, https://docs.openstack.org/developer/keystone/

[19] The OpenStack Foundation, 2017, *Identity API v3* https://developer.openstack.org/api-ref/identity/v3/

[20] Cyril Dangerville, 2017, Authorization PDP - AuthZforce, https://catalogue.fiware.org/enablers/authorization-pdp-authzforce

[21] Erik Rissanen (Ed.), 2013, eXtensible Access Control Markup Language (XACML) Version 3.0, http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html

[22] Álvaro Alonso et al, 2018, PEP Proxy – Wilma, http://fiware-pep-proxy.readthedocs.io/en/latest/

[23] WireCloud Team, 2018, Application Mashup - Wirecloud https://catalogue-server.fiware.org/enablers/application-mashup-wirecloud

[24] Álvaro Arranz et al, 2018, Widget & Operator Development, https://wirecloud.readthedocs.io/en/latest/development/widget_and_operators/

[25] Álvaro Arranz et al, 2018, *Wirecloud User-Guide*, https://wirecloud.readthedocs.io/en/stable/user_guide/

[26] T. Dierks, 2008, *The Transport Layer Security (TLS) Protocol, Version 1.2*, Internet Engineering Task Force – Network Working Group, RFC 5246, https://tools.ietf.org/html/rfc5246

[27] E. Rosen, Y. Rekhter, 1999, *BGP/MPLS VPNs*, Internet Engineering Task Force – Network Working Group, http://www.ietf.org/rfc/rfc2547.txt